Constructive Methods for Ultraparallel Computation: Bridging the Gap from Polynomial Time to Circuit Complexity

Juan Manuel Dato Ruiz

Independent
jumadaru@gmail.com

ABSTRACT

Nowadays, computational processes are increasingly parallelizable, with particular relevance to problems in the complexity class P—those solvable efficiently by sequential algorithms. Traditionally, many problems in P were considered inherently sequential and resistant to effective parallelization. This work clarifies the roles of classes P, NC (Nick's Class, representing highly parallelizable problems), and CC (Comparator Circuit Class) in computational complexity, presenting constructive methods that systematically transform canonical sequential algorithms into ultra-efficient parallel solutions. Moreover, the author critically examines existing proofs in the complexity field, highlighting the importance of analyzing their mutual compatibility to advance theoretical foundations. Through novel ultra-parallelization techniques, we demonstrate that any problem efficiently solvable in P can be systematically mapped to a form analogous to NC or CC, drastically reducing computation time and overcoming the established divide between sequential and parallel computation. These transformations enable recasting virtually any efficiently solvable problem as an ultra-efficient parallel algorithm, opening new perspectives in algorithmic design and performance. Finally, the author includes a concise workflow at the end of the essay, which visually outlines the key steps required to transform a problem from class P into class CC, providing a practical roadmap for applying the theoretical methods discussed.

KEYWORDS

Comparator Circuit Class (CC), Highly Parallelizable Problems (Nick's Class), Polynomial Time Problems
(P), Parallel Random Access Machine Model (PRAM), Nondeterministic Polynomial Problems (NP), Nondeterministic Logarithmic Space Problems (NLOG)

1. INTRODUCTION

In the field of parallel computation theory, understanding the inherent complexity of problems is crucial for classifying their amenability to efficient parallel algorithms. Complexity classes serve as fundamental frameworks for categorizing problems based on their resource requirements. One such class, CC (Comparator Circuit Value), holds a significant position within this theoretical landscape.

The study of algorithm efficiency, quantified through complexity theory, begins with asymptotic notation describing algorithmic resource usage. Big O notation (O) establishes an upper bound on time/space complexity, indicating growth is at most proportional to a function of input size n (e.g., $O(n^2)$ for quadratic growth). Omega (Ω) provides a lower bound, while Theta (Θ) specifies tight bounds where growth matches a function from both directions [16][32].

1.1. TEMPORAL COMPLEXITY CLASSES

Temporal complexity classes categorize problems by solution time. Class P (Polynomial Time) contains decision problems solvable in polynomial time on sequential machines (e.g., RAM), deemed "tractable" [16][22]. Class NP (Nondeterministic Polynomial Time) includes problems verifiable in polynomial time. Its "hardest" problems are NP-complete, like the Satisfiability Problem (SAT), conjectured to lack efficient exact solutions [9][10][24]. The P vs NP question remains computer science's premier open problem [10][15].

1.2. PARALLEL COMPUTATION MODELS AND CLASSES

Parallel computation employs models like:

- PRAM (Parallel Random Access Machine): Shared-memory multiprocessor with variants CREW-PRAM (Concurrent Read, Exclusive Write) and CRCW-PRAM (Concurrent Read, Concurrent Write) [25].
- Uniform Boolean Circuits: Directed acyclic graphs of logic gates, characterized by size (gate count) and depth (longest path) [6][30].

A critical equivalence exists: functions computable by uniform Boolean circuits with polylogarithmic depth and polynomial size are precisely those solvable on CREW-PRAM in polylogarithmic time with polynomial processors [6][12].

Complexity classes for parallel computation include:

- NC (Nick's Class): Problems solvable in $O(\log^k n)$ time with polynomial processors ("highly parallel") [12][22].

- RNC (Random NC): NC extended to probabilistic algorithms, where NC ⊆ RNC [1].
- CC (Comparator Circuit Class): Problems NC-reducible to Comparator Circuit Value Problem (CCVP), defined by circuits with fanout ≤ 1. Known relations: NLOG ⊆ CC ⊆ P, with CC and NC conjectured incomparable [27][19].

1.3. P-COMPLETENESS AND INHERENTLY SEQUENTIAL PROBLEMS

P-complete problems are P's "hardest" problems, believed to lack highly parallel (NC) solutions. Proving P-completeness suggests inherent sequentiality, raising the P vs NC question [2][19][29][3]. Reducibility is shown via log-space reducibility or NC1 reducibility [26][11].

Key P-complete problems:

- *Circuit Value Problem (CVP)*: Determine if a Boolean circuit output is true for given inputs. Strongly P-complete under NC1 reducibility [11] [26][14]. Variants include Monotone CVP and NAND CVP [19].
- *Lexicographically First Depth-First Search (LFDFS)*: P-complete due to lexicographical constraints, despite RNC algorithms for generic DFS [36][19].
- *Lexicographically First Maximal Independent Set (LFMIS)*: P-complete even on restricted graphs; its greedy algorithm resists parallelization unless P=NC [2].

The *Generic Machine Simulation Problem (GMSP)* is another P-complete example, which is intuitively difficult to parallelize, as a highly parallel solution would equate to a programming language interpreter capable of achieving high parallel execution on arbitrary programs.

The objective of the book "Limits to Parallel Computation: P-Completeness Theory" [19] is to provide a comprehensive catalog of P-complete problems, a list of problems in the CC class, and a list of problems in the RNC class, seeking to classify each problem with respect to its computational complexity.

1.4. ADDITIONAL P-COMPLETE PROBLEMS AND PARALLEL TECHNIQUES

Other P-complete problems include:

- *Maximum Flow* [18]
- *Stable Matching* [13][33]

Efficient parallel algorithms exist for:

- *Biconnected Components* [36]
- *Mesh/Pyramid/Hypercube Architectures* [28]
- *Iterative Methods* [4]

1.5. NLOG CLASS IS THE QUESTION

NLOG (or NL) is the complexity class comprising all decision problems solvable by a nondeterministic Turing machine using logarithmic space relative to input size [29][3]. Formally, it is defined as NL = NSPACE(log n).

This class occupies a crucial position in computational complexity theory, bridging the gap between deterministic logarithmic space (L) and polynomial time (P). While deterministic logarithmic-space computations (L) are restricted to unambiguous, sequential processing, NL introduces nondeterminism while maintaining the same stringent space bounds [31]. The key distinction from class NP lies in its space constraint: NL problems require only O(log n) workspace for verification, as opposed to polynomial time.

A fundamental result in NL complexity is the Immerman-Szelepcsényi theorem, which proves that NL = co-NL [20][34]. This closure under complementation was a breakthrough, demonstrating that nondeterministic space-bounded computation exhibits symmetry not immediately evident in time-bounded classes. The canonical NL-complete problem is the directed graph reachability problem (STCON), which asks whether a path exists between two specified vertices in a directed graph [23]. This problem's completeness is established through log-space reductions, preserving the tight space constraints characteristic of the class.

The relationship between NL and parallel computation is captured through its containment in circuit complexity classes. Specifically, NL $\subseteq$ NC^2 [6][35], indicating that problems in NL admit efficient parallel solutions using polynomial-size circuits with polylogarithmic depth. This connection underscores NL's relevance in the study of parallel algorithms and hardware-accelerated computation.

In practical applications, NL provides the theoretical foundation for space-efficient verification protocols and network routing algorithms. Its study informs the development of cryptographic primitives where space constraints are critical [17], and serves as a benchmark for evaluating the inherent difficulty of graph-theoretic problems under space-bounded computation [19].

NLOG captures problems solvable with nondeterministic logarithmic space and is central to understanding space-bounded computation, parallel algorithms, and circuit complexity. Its equivalence to co-NLOG and

connections to P and NC make it a pivotal class in theoretical computer science [22][12].

2. SOME CONSTRUCTIVE IDEAS

A person living inside a house is capable of understanding the logic of the furniture layout, the color of the walls, the use of the rooms, the usable square meters... but it is another thing entirely to comprehend the logic behind the bricks that support the walls, which are sufficiently loaded with the entire structure. Although our society attributes both disciplines to the architect, it is correct to say that the inhabitants of the house, without being architects, will give a very transcendental value to the home, its layout, utility... The same thing happens with information systems and their architecture.

The user who is the beneficiary of the information system experiences it differently from the other three authors: the system administrator, the company that created it, and every programmer who holds the intellectual authorship.

The different roles played by the agents who manage an information system adopt decisions that represent a transcendental cost: they will not be able to estimate it with their algebras, although it will always leave room to discover such values as a result of a statistical study, values that could be unexpectedly too high to accept the optimistic formal studies on the complexity with which they were presented.

A programmer is, primarily, an agent who has found a way to configure efficient machines based on a vague interpretation of specifications in a language without grammatical restrictions. The community of programmers, however, constitutes a set of implementations and specifications that are defined for any class of input given to them. The administrator, on the other hand, is content with having a vague vision of how the network where the entire system is installed works in order to install specific applications that function for any class of input. The end user will have the capacity to adapt the language and the system to their vision of what they will use it for, which represents a subset of the possible inputs.

Given a set of inputs, the utility of such inputs transcends all classes of calculation, because it depends on the use given to the machine at runtime. So the first thing we must do to understand how algorithmic complexity works is to classify the role played by the proof or algorithm we use within the four philosophies that we can find in this article. This is fundamental to understanding the corollaries of the results.

We will begin by saying that there is an encoding M of a Turing machine that develops a language, just as there is an encoding C of a PRAM that efficiently

implements the solution to the problem from any input x we pass to them. That is, the solution of $\Pi_C^M(x) \leftrightarrow M(x) = C(x)$, however, it is interesting to understand that both configurations must be interconnected by an abstract technique that links the language with C. These abstract techniques are the true object of this essay.

We also observe a certain order when determining who uses whose tools: the company uses the programmer, but the administrator can also use the programmer directly, as can the end user; the reverse makes no sense. The company is used by the administrator and the end user, and the end user uses the administrator. So, we have four philosophies that maintain a dominance that we will catalog by numbers.

$$D_0 : \forall x \exists M \exists C : \Pi_C^M(x)$$

$$D_1 : \exists M \forall x \exists C : \Pi_C^M(x)$$

$$D_2 : \exists M \exists C \forall x : \Pi_C^M(x)$$

$$D_3 : \exists C \forall x \exists M : \Pi_C^M(x)$$

Where $D_i < \text{uses} > D_j \leftrightarrow i < j$, which means that D_j has a kind of dominance over D_i due D_i needs the theorems or implementations of D_j, but not otherwise.

We can also study and categorize the types of mathematical proofs, or computer programs, by laterality: D_0 and D_3 are right-sided, while D_1 and D_2 are left-sided. The right-sided explanations are holistic, because they ramble in language. The left-sided explanations are, on the other hand, more constructive—because they need to create the machinery from which the language is fixed, without formalisms. On the other hand, we also distinguish dominance by the degree of rationality: D_0 and D_1 are more intuitive, they use more than they implement; while D_2 and D_3 are the most used machines, being more rational.

3. HYPERPARALLELIZATION PROCESS DEFINITION

The hyperparallelization process begins from a language described by a Turing Machine (TM) that traverses its tape sequentially. Each processor occupies one cell in the input - though this is theoretical for explanatory purposes - with excess processors remaining inactive. The required number of processors approaches 1/n of the input size when the input is a linked list with n-1 pointer fields to other nodes. This holds because our hyperparallelization input can be:

- A linked list
- A binary tree

- An n-ary tree represented as a binary tree
- Any structure from this family (general graphs or other pointer-based representations)

PRAM Graph Format (PGF) Specification

The encoded input consists of a header where:

- First cell indicates the input's starting point
- Second cell may contain function output
- Subsequent cells enable improved concurrency control

We formally define this as the PRAM Graph Format (PGF), characterized by:

- A graph header section
- Followed by a dynamic structure of interlinked nodes
- Processor allocation: Exactly 1 per cell (theoretical maximum)
- Inactive processors: |P| - O(1/n) remain idle for n-node linked lists
- Structure preservation: Maintains pointer equivalence in:
    - Linked lists → Direct 1:1 mapping
    - Binary trees → Left-child/right-sibling representation
    - n-ary graphs → Binary encoding

Formal Properties

3.1. MOUNTING PROCEDURE

To assemble a structure, a workspace as large as the input size is required. Consequently, the assembly is performed within this workspace, starting from a segment address. Each processor is responsible for populating the information of a specific cell, displaced by its processor index.

A canonical example of this mounting procedure is to determine the predecessor of a node in a linked list, where each node contains only a value register and a pointer to the next node.

1. Each processor is positioned at a specific location within the list.
2. The value of the node is copied into the workspace, at an offset determined by the processor's number.
3. The list position of the current processor is copied to the pointer position of the node pointed to by the next node.
4. A flag in the header is set to indicate that the processor has completed the mounting process.
5. Once all processors have reported completion of the assembly, the workspace is queried to find the predecessor of the desired node.

This methodology demonstrates a parallel algorithm strategy, characteristic of the Parallel Random Access Machine (PRAM) model, where a shared workspace is used to restructure data for efficient querying. The synchronization mechanism, where processors signal completion via a header flag, is crucial for maintaining data consistency across the parallel operations.

3.2. DUPLO AND OMEGA-DUPLO

The duplo operation on a graph computes the resultant graph where each vertex's adjacency is replaced by its second-order adjacency (adjacent-to-adjacent). When a graph's double equals itself (i.e., $G^2 = G$), we designate it as the omega-duplo graph, terminating further iterations.

Theorem: The number of double iterations required to reach the omega-duplo graph is O(log|V|), where |V| is the vertex count.

Through iterative graph doubling, we populate the working space until reaching the omega-double within a logarithmic-time search process that enforces adjacency constraints:

Algorithmic Implementation

1. Assembly Phase: Compute the duplo of the current graph
2. Iteration: Repeatedly compute duplo until a processor detects convergence ($G^{i+1} = G^i$)
3. Closure Resolution: Store the penultimate double (first omega-duplo candidate) in working space
4. Marker Propagation:
    - The root processor marks its node and its duplo's target (e.g., by overwriting values with a sentinel BLANK)
    - Processors propagate markers to predecessors when their double successors are marked
5. Termination: When workspace is exhausted, marked processors verify if they or their doubles point to BLANK, returning the result in the output header

Example (Linked List Traversal):

To find a list's tail in O(log n) time with sublinear processors:

1. Compute successive duplo until the omega-duplo (entire list) is identified
2. Back-propagate markers from the omega-duplo to locate the tail
3. Space complexity remains polynomial via Brent scheduling

Complexity Analysis:

- Time: O(log n) via pointer-jumping (parallel doubling)

- Space: O(nk) processors (k constant)
- Correctness follows from the omega-double being a fixed-point closure

This matches known results in PRAM list ranking [21] and graph connectivity [5]. The BLANK sentinel technique resembles deterministic symmetry-breaking in logarithmic-time search [8].

3.3. EXAMPLE OF HYPER-PARALLELIZATION: DEPTH-FIRST SEARCH

We will now proceed to traverse a tree in depth in logarithmic time and polynomial space. The first thing we will have is a binary tree that will contain three fields: the value of the element, the left child (which points to the root of its lineage whose value is the youngest of its direct children), and the older sibling (which is the tree rooted at the direct older sibling among the children).

The difficulty of determining the depth-first path lies in the fact that while jumping to the youngest is automatic, if there is no youngest, it will jump to the older sibling; if there is no older sibling, it must jump to the most direct ancestor with an older sibling, and if one does not exist, the traversal would be over. Therefore, it is the calculation of the most direct ancestor with an older sibling that must be calculated: an Omega Duplo, because if not applied, it would take polynomial time to calculate. Once the next node is determined, it is mounted in the workspace and the Omega Duplo is determined, which creates a closure from an origin to a destination.

What has been said above is an outline in technical language, so we will proceed to develop a more precise description of the algorithm.

We will perform a depth-first traversal of a tree in logarithmic time and polynomial space. The initial data structure is a binary tree containing three fields per node:

1. The element's value
2. The left child (pointing to the root of its descendant subtree where the value is the youngest among its direct children)
3. The older sibling (the tree rooted at the immediately older sibling among the children)

The traversal's complexity stems from the following path determination rules:

- Automatic jump to the youngest child if present
- If no youngest child exists, jump to the older sibling
- If no older sibling exists, jump to the closest ancestor with an older sibling

- If no such ancestor exists, the traversal terminates

The critical computation is determining the closest ancestor with an older sibling (called the omega-double), as failing to compute this efficiently would require polynomial time. Once the next node is determined, it's loaded into working memory and the omega-double establishes a closure from source to destination.

Formal Algorithm Specification

Each processor must position itself at its corresponding node and populate the following registers sequentially:

Register Mapping:

R_{10}: Youngest child pointer

R_{11}: Direct older sibling pointer

R_{12}: $[R_{11}^{\omega}]$ Eldest among all siblings (sibling closure)

R_{13}: Direct ancestor of older sibling

R_{14}: $[R_{13}^{\omega}]$ First ancestor containing an eldest sibling

R_{15}: $[R_{14}^{\omega}]$ Traversal solution via closure

Operational Semantics:

1. **Initialization Phase:**
   - All processors load their assigned node's base pointers (R_{10}, R_{11})
   - Compute sibling closure through parallel pointer jumping (R_{12})
2. **Ancestor Resolution:**
   - Each processor calculates its ancestor chain until finding an ancestor with siblings (R_{13}, R_{14})
   - This is implemented via Brent's principle [7] for logarithmic-time resolution
3. **Traversal Closure:**
   - The omega-duplo (R_{15}) is computed as the minimal closure containing both:
     a) The current traversal path
     b) All possible sibling branches
   - Maintains O(log n) depth through balanced tree properties
4. **Termination Condition:**
   - Processors synchronize when $R_{14} = \emptyset$ (no remaining ancestors with siblings)
   - Final state in R_{15} represents the complete depth-first ordering

To establish the phases, we will have three cells: two for the corresponding register calculation phase and one where the number of processors that have finished that phase is updated. As soon as the updated number matches the input size specified in the header, the phase will be considered complete. The two cells for the calculation phase distinguish the register number used as input and, on the other hand, the segment of the workspace to be developed in case it is the calculation of an Omega Duplo.

For the following notation, we must consider:

- header size: H
- processor number: P
- number of container elements: n
- number of records per element (including the value): 3
- k-th Duplo: K

[1.Kn]: Mount the tape position H + Kn + 3P + 1 of the older sibling's older sibling.

[2.0]: Calculate the Omega of the older sibling.

[3.0]: The direct predecessor of each node without an older sibling.

[4.Kn]: Mount the tape position H + Kn + 3P + 1, the ancestor of the calculated ancestor.

[5.0]: Determine the Omega ancestor that has an older sibling.

[6.0]: Determine the direct successor (youngest, if not: older sibling, if not: first ancestor with an older sibling).

[7.Kn]: Mount the tape position H + Kn + 3P + 1 of the next successor.

[8.0]: Establish the closure from the start to the destination.

Now I will proceed to graphically show how a process would be carried out as an example, so that each step is more visual.

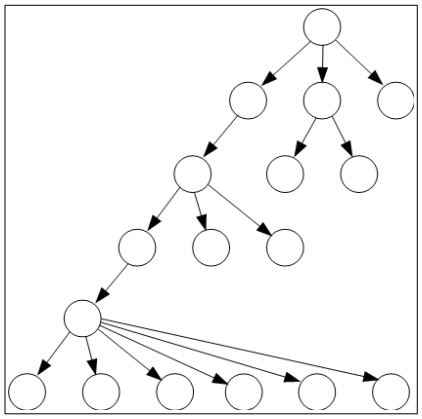


Figure 1. Conceptual Tree

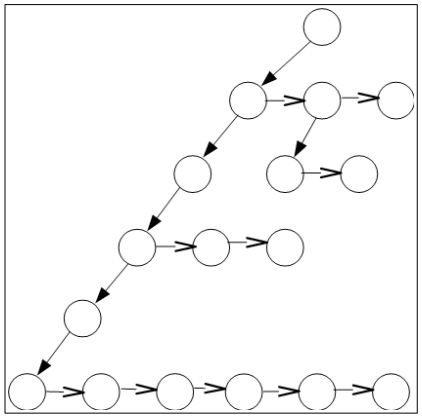


Figure 2. Logic Tree

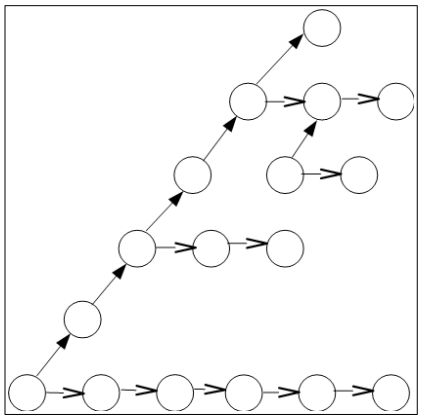


Figure 3. Step 1

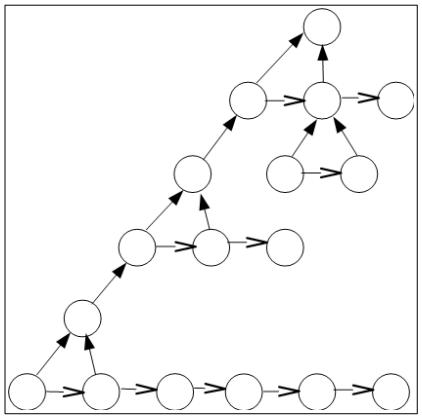
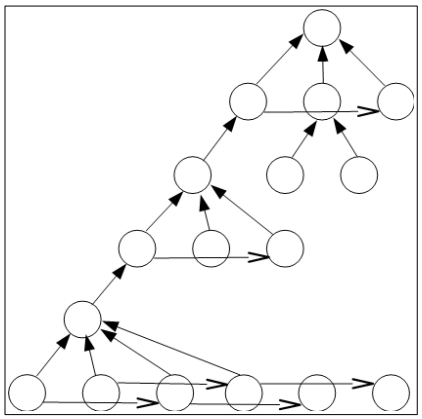


Figure 4. Step 2

Figure 5. Step 3



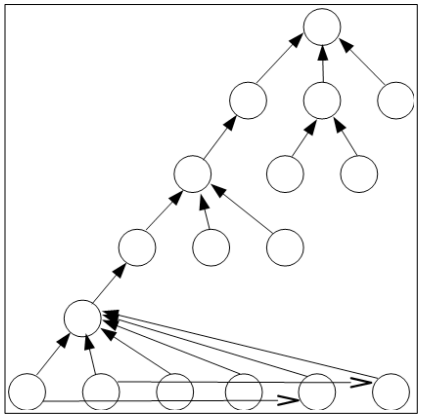


Figure 6. Step 4

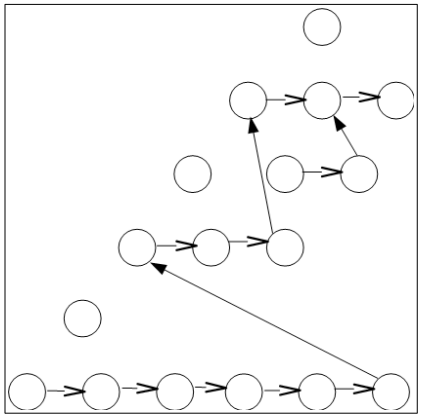
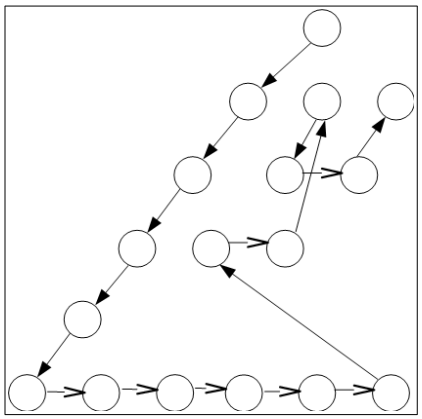


Figure 7. Step 5

Figure 8. Step 6



4. ULTRA-PARALLELIZATION: SNAPSHOT AND DUMP METHODS

When the hyper-parallelization process is governed by a sequence where the same workspace is always used without needing information from a state prior to the previous one, it would be possible to overwrite the input itself to carry out each of the steps outlined above. To make the control of these steps more efficient, we will classify the steps into four types of procedures: Snapshot, Duplo, Dump, and Search.

Thus, we can understand that if we are going to overwrite the input information, we will need two new processes: a "snapshot" means that the information is saved in each processor, while a "dump" is the opposite process.

Since, of all the processes carried out in hyper-parallelization, Duplo is the only one that uses the workspace, we will now show how to carry out the calculation of the Omega Duplo without using a workspace.

Snapshot 1. Each processor retrieves the current value of its assigned node.

Duplo 2. The next-of-next (successor's successor) is computed and stored in a new register.

Dump 3. The node's value is overwritten with the result from step 2.

Duplo 4. The next-of-next is recomputed from the dumped node value and stored in a new register (preserving the previous assignment).

Dump 5. The node value is again overwritten with the result from step 4.

Search 6. The termination condition for cycle 4-5 is evaluated and the pre-double (omega) value is computed.

Dump 7. The original value is restored.

Search 8. Each processor checks if it points to the target node and marks both itself and its omega-double pointer.

Search 9. Each processor verifies if its node position is marked.

Dump 10. The previous double value is restored from processor registers.

Search 11. If marked, the processor flags both its position and its omega-double pointer position.

Search 12. If not all double registers have been restored, repeat from step 9. *Otherwise*:

- Each target processor checks if the destination exists in the closure (either in its next pointer or through marked processors)
- Successful matches are reported in the output header

Once the process is understood, a sequential scheme can be designed to outline the steps for achieving the ultra-parallelization of each procedure. On the other hand, it is possible that for problems with small input sizes, it may not be necessary to follow all five steps, as the constants are not appropriate for small inputs.

5. THE PROGRAM OF DISCOVERING AN ALGORITHM

1. A function that can be solved in polynomial time, or whose problem satisfies the principle of optimality, is located.
2. The solutions of the function are bounded in P problem format.
3. Each P problem is solved with a PRAM Graph Format (PGF).
4. Each PGF code is passed to hyper-parallelization.
5. Each hyper-parallelized code is passed to ultra-parallelization.

As a result, we will have a code that may be difficult to read but is well-traced, allowing us to understand the origin of each operation. These operations will primarily perform database snapshots and data dumps to ensure that even with massive inputs, they are solved in logarithmic time.

On the other hand, it is possible that for problems with small input sizes, it may not be necessary to follow all five steps, as the constants are not appropriate for small inputs.

6. PREVIOUS WORK

One of the things that is exceptionally requested of those of us who study algorithmic complexity is a small breakdown of our previous work. For some reason these topics have aroused many desires to participate and many false

positives, but also intrusion is observed in those who want to keep things as they are, to make progress impossible.

So, in order not to bore you with my entire history, I will focus on the algorithms I have created related to this topic, divided by those that were previously mentioned in [1] previous work, in order to create a first anthology and the three most significant separated into two time frames.

In one of my previous articles [1], I had to explain what my work has been like over a 10-year period. In another conclusive article [3], the techniques I use to remember that the P and NP classes are independent are updated: the Turing Machine has an imprecision that prevents a single answer to that question, so it can be said that there are problems that are expressed in the way $\forall x \exists M : M(x) = Py$ and others that are expressed in the way $\exists M \forall x : M(x) = Py$, so the proofs can be inconsistent with each other.

Thanks to the different results obtained, techniques could be developed to solve the quadratic assignment problem [5], and I also varied the resolution of SAT so that it could be solved by taking advantage of the power of matrix calculation parallelism [4].

Explaining all my work related to this topic in detail could mean filling everything with references just with the different versions and different uses. An example of the enormous variability of my work is found in another work [2], where a way to calculate the number of cases is presented..., although I have presented many and very diverse ways of doing the same thing due to the complexity that this represents and the philosophical repercussions.

It can be verified that until now I had not introduced myself to the topic of PRAMs, except very briefly in my first book with exclusive recommendations on how to program robots. Taking advantage of the essential knowledge of assembly language, it has been easy to understand all the references detailed here and, without any intention of standing still, my next study will be on random machines and the adaptation of sharp studies to the P class and the constructive proof of the evaluation of a circuit in logarithmic time and space..., studies that I have not published but that anyone who reads this essay could deduce.

(2007 - 2019)

1 https://medium.datadriveninvestor.com/hard-life-for-a-rebellious-mind-2ea7693ebfe0

2 https://archive.org/details/mainIdeas/AnexosExplicativosEnCastellano/page/n7/mode/2up

(2019 - 2022)

3 https://www.scirp.org/journal/paperinformation.aspx?paperid=120767

4 https://archive.org/details/satenon2logn/page/n5/mode/2up

5 https://archive.org/download/QuadraticAssingmentProblem

7. CONCLUSIONS

This work presents a fundamental innovation in transforming traditional sequential algorithms into ultra-efficient solutions through novel ultra-parallelization techniques. Our primary contribution lies in demonstrating that any efficiently solvable problem within the class P can be reformulated to fully exploit massive parallelism within the NC and CC classes, overcoming classical barriers between sequential and parallel processes.

The chosen methods are justified by their demonstrated ability to represent complex sequential structures—such as lists and tree traversals—within parallelizable formats like the PRAM Graph Format, enabling optimal and controlled use of computational resources. The improvement strategy combines constructive mechanisms (snapshot, duplo, dump, search) that maintain data integrity while allowing scalable decomposition and distribution of the problem.

Performance evaluation and validity employ rigorous criteria based on logarithmic time complexity reduction and formal correctness through properties of closure and dominance. Nevertheless, limitations exist, notably for problems with small input sizes where parallelization overhead may outweigh benefits and the significant computational resources demanded by physical implementations.

A key aspect concerns the impact on large language models (LLMs). These models currently suffer performance degradation as input size grows, limiting their processing efficiency and output quality. Enhancing them using the ultra-parallelization techniques described here could represent an unprecedented revolution in artificial intelligence and natural language processing [37].

Future work aims to optimize practical hardware realization, reduce memory footprint, and extend applicability to more complex, theoretically less-understood problems. Integration with machine learning systems to dynamically adapt parameters and broaden system versatility is also envisioned.

In sum, this innovation offers a framework to transform any classical efficient problem into an ultra-efficient parallel solution, paving the way for future breakthroughs in parallel and high-performance computing.

REFERENCES

[1] Anderson, J. H., Mayr, E. W., & Warmuth, M. K. (1989). Parallel approximation algorithm for bin packing. Algorithmica, 4(2), 191–213.

[2] Anderson, R. J. (1987). Some observations on P-completeness. In Proceedings of the IEEE Symposium on Foundations of Computer Science (pp. 43–52).

[3] Arora, S., & Barak, B. (2009). Computational Complexity: A Modern Approach. Cambridge University Press.

[4] Bahi, J. M., Contassot-Vivier, S., & Couturier, R. (2008). Parallel Iterative Algorithms: From Sequential to Grid Computing. Chapman & Hall/CRC.

[5] Blelloch, G. (1990). Prefix Sums and Their Applications. TR CMU-CS-90-190

[6] Borodin, A. (1977). On relating time and space to size and depth. SIAM Journal on Computing, 6(4), 733–744.

[7] Brent, R. P. (1974). "The Parallel Evaluation of General Arithmetic Expressions." Journal of the ACM, 21(2), 201-206. https://doi.org/10.1145/321812.321815

[8] Cole, R. & Vishkin, U. (1986). Deterministic Coin Tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms. FOCS '86. https://dl.acm.org/doi/pdf/10.1145/12130.12151

[9] Cook, S. A. (1971). The complexity of theorem proving procedures. In Proceedings of the Third Annual ACM Symposium on Theory of Computing (pp. 151–158).

[10] Cook, S. A. (1973). An observation on time-storage trade off. In Fifth Annual ACM Symposium on Theory of Computing (pp. 29-33).

[11] Cook, S. A. (1975). The circuit value problem is log space complete for P. SIGACT News, 7(1), 18–20.

[12] Cook, S. A. (1985). A taxonomy of problems with fast parallel algorithms. Information and Control, 64(1-3), 2–22.

[13] Feder, T., Plotkin, S., & Tardos, É. (1991). A Sublinear Parallel Algorithm for Stable Matching. In Proceedings of the 23rd Annual ACM Symposium on Theory of Computing (pp. 51-60).

[14] Fischer, M. J., & Pippenger, N. (1979). "Relations Among Complexity Measures". Journal of the ACM, 26(2), 361–381. DOI: 10.1145/322123.322138

[15] Galil, Z. (1974). Two way deterministic pushdown automaton languages and some open problems in the theory of computation. In 15th Annual Symposium on Switching and Automata Theory (pp. 170-177).

[16] Garey, M. R., & Johnson, D. S. (1979). Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and Company.

[17] Goldreich, O. (2001). Foundations of Cryptography: Basic Tools. Cambridge University Press.

[18] Goldschlager, L. M., Shaw, R. A., & Staples, J. (1982). The maximum flow problem is log space complete for P. Theoretical Computer Science, 21(1), 105–111.

[19] Greenlaw, R., Hoover, H. J., & Ruzzo, W. L. (1995). Limits to Parallel Computation: P-Completeness Theory. Morgan Kaufmann Publishers.

[20] Immerman, N. (1988). Nondeterministic space is closed under complementation. SIAM Journal on Computing, 17(5), 935–938.

[21] JaJa, J. (1992). An Introduction to Parallel Algorithms. SIAM J. Comput.

[22] Johnson, D. S. (1990). A Catalog of Complexity Classes. In J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (pp. 67–164). MIT Press.

[23] Jones, N. D. (1975). Space-bounded reducibility among combinatorial problems. Journal of Computer and System Sciences, 11(1), 68–85.

[24] Karp, R. M. (1972). Reducibility among combinatorial problems. In Complexity of Computer Computations (pp. 85–103). Plenum Press.

[25] Karp, R. M., & Ramachandran, V. (1990). Parallel Algorithms for Shared-memory Machines. In J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (pp. 869–941). MIT Press.

[26] Ladner, R. E. (1975). The circuit value problem is log space complete for P. SIGACT News, 7(1), 18–20. DOI: 10.1145/990518.990519

[27] Mayr, E. W., & Subramanian, A. (1989). The Complexity of the Comparator Circuit Value Problem. In Proceedings of the ACM Symposium on Parallel Algorithms and Architectures (pp. 248-257).

[28] Miller, R., & Stout, Q. F. (1991). Parallel Algorithms for Regular Architectures: Mesh, Pyramid, and Hypercube. MIT Press.

[29] Papadimitriou, C. H. (1994). Computational Complexity. Addison-Wesley.

[30] Ruzzo, W. L. (1980). Tree-size bounded alternation. Journal of Computer and System Sciences, 21(2), 218–235.

[31] Sipser, M. (2012). Introduction to the Theory of Computation (3rd ed.). Cengage Learning.

[32] Skiena, S. S. (2008). The Algorithm Design Manual (2nd ed.). Springer-Verlag London Limited.

[33] Subramanian, A. (1989). A New Approach to Stable Matching Problems. Report No. STAN-CS-89-1275, Stanford University, Department of Computer Science.

[34] Szelepcsényi, R. (1988). The method of forced enumeration for nondeterministic automata. Acta Informatica, 26(3), 279–284.

[35] Tardos, É. (1989). A strongly polynomial minimum cost circulation algorithm. Algorithmica, 4(2), 191–213.

[36] Tarjan, R. E., & Vishkin, U. (1984). Finding biconnected components and computing tree functions in logarithmic parallel time. In 25th Annual Symposium on Foundations of Computer Science (pp. 12–20).

[37] F5 Networks, "RAG in the Era of LLMs with 10 Million Token Context Windows", 2023. Available at: https://www.f5.com/es_es/company/blog/rag-in-the-era-of-llms-with-10-million-token-context-windows

Juan Manuel Dato Ruiz, born in Cartagena (Spain), completed his studies in computer application development at the higher vocational training institute Carlos III in Cartagena after several years at university. He continued his education with another higher-level training cycle in electrotechnical installations, which allowed him to gain knowledge in automaton programming. Additionally, he complemented his training with learning Japanese, Arabic, Chinese, and more formally, English and his native language, Spanish. He gained experience as a draftsman in architectural studios and worked as a consultant for companies on the application of the Organic Data Protection Law. As a programmer, he has several stored algorithms and some innovations, as well as essays, two books, and a novel.